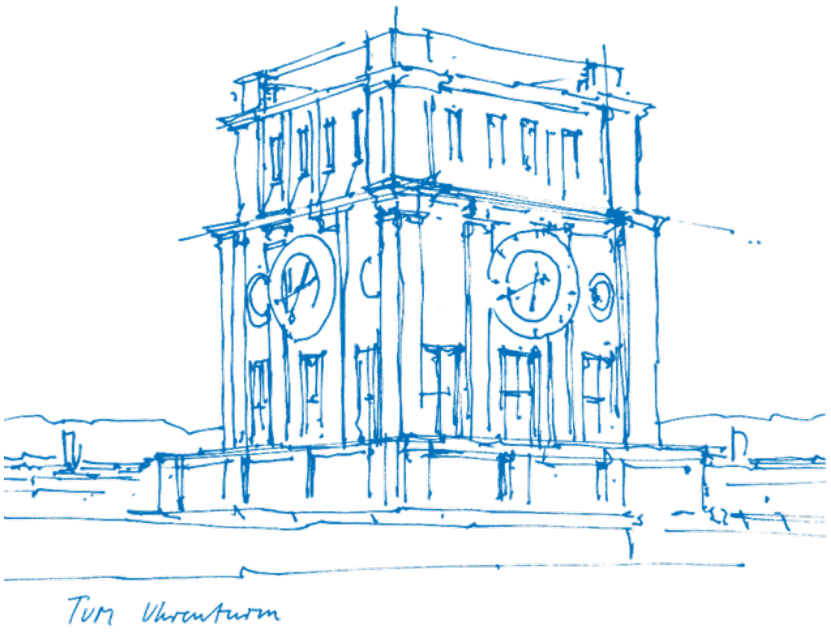
TUM Department of Electrical and Computer Engineering
Chair of Security in Information Technology
Technical University of Munich

Analysis of Machine Learning for State Register Identification

Lee Seng Hwee
Bachelor of Science in Electrical Engineering and Information Technology

Bachelor's Thesis
Bachelor of Science
Department of Electrical and Computer Engineering
Technical University of Munich

Supervised by
Prof. Dr.-Ing. Georg Sigl
M.Sc. Johanna Baehr
Chair of Security in Information Technology



I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Contents

1 Introduction

1.1 Hardware Vulnerability

In this fast-moving world, no one company can innovate and manufacture all minute components and modules itself. This is especially so in the designing and manufacturing of highly complex chips. Hence there is a strong reliance on third party solutions. With an increase in this reliance, there is an increase in opportunities for hardware security to be compromised, especially during the fabrication process, whereby additional backdoors can be implemented by the manufacturer. These backdoors which are not known to the designers can be exploited leading to loss of privacy, data and potentially system failures. In most cases this back-door can become hardware vulnerabilities which cannot be patched or takes time to be patched via a software update. Therefore, at those times, hardware might have to be replaced which can incur high amounts of cost and brand names associated with those hardwares, resulting in a detrimental decline of consumer trust.

1.2 Reverse Engineering Process

To mitigate the risk of hardware vulnerabilities caused by third party manufacturers, there is therefore a need to verify the manufactured chips especially if the manufacturer is not reputable. Hardware reverse engineering methods have been developed more in recent years to detect these security compromises, given the significant increase in complexity of chip designs. However traditional reverse engineering is still a tedious process as shown in Figure 1.1, involving depackaging, delayering, using of advanced imaging techniques to capture each layer and further processing to extract the netlist, which is explained in more detailed in paper[1].

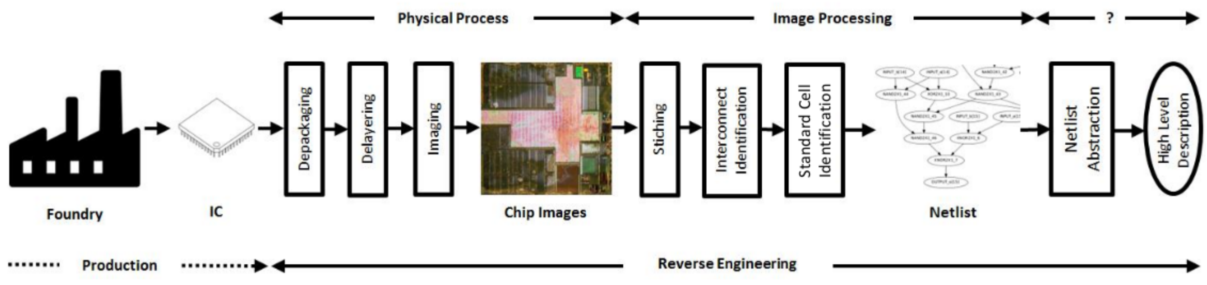


Figure 1.1 Reverse Engineering Workflow[1]

1.3 Severity of Hardware Vulnerability

To emphasize the severity of hardware vulnerability, past examples of hardware vulnerability, not specifically caused by manufacturers who added in back-door, can be used to elaborate this. "Meltdown"[2] and "Spectre"[3] are 2 recent high profile cases of hardware vulnerabilities, which employed a similar form of attack on the processor, by targeting the isolation between the application and operating system and the isolation between applications respectively. This resulted in leakage of sensitive information. Luckily, these vulnerabilities could be patched via software patches.

1.4 Current Approaches

As explained in section 1.2, reverse engineering is a tedious process, and this is especially true when dealing with State Register Identification from an extracted netlist. The current approach in identifying State Registers, requires designers with the full design information to compare the original design to reverse engineered design to identify if any additional gates or logic has been implemented. However, this method does possess limitations, commercial-of-the-shelf (COTS) ICs and SOCs with third-party IPs lack a golden model to be compared to.

1.4.1 RELIC and fastRELIC

To overcome the lack of a golden model, RELIC[4] and fastRELIC[5] were developed. RELIC and fastRELIC both use similar methods of finding potential state registers based on register pairs similarity scores (PSS). RELIC first pre-processes a gate level netlist to guarantee a similar gate structure, the State Registers from the netlist can be then separated out from the Data Registers, because the structure and functionality surrounding them is inherently different to the Data Registers, where the same functionality will happen more than once, i.e. for a 32 bit calculation, a function will occur 32 times. Comparing RELIC and fastRELIC, fastRELIC differs slightly by using a grouping mechanism, based on some parameters while RELIC chooses a specific parameter and declares any state registers with a lower cumulated PSS than that to be a State Register. This approach, though results high in accuracy, only rely on the PSS generated.

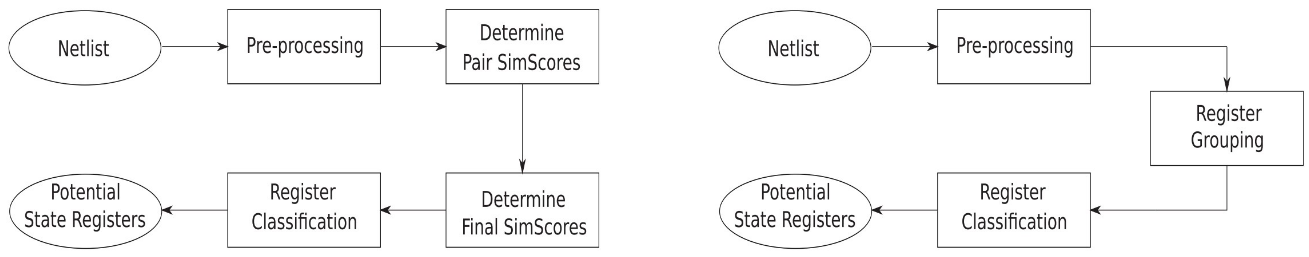


Figure 1.2 RELIC and fastRELIC

The aim of this thesis is to train and deploy a neural network, using more features from the extracted netlist, aiding the decision making of classifying a Register into the categories of either 1 or 0 with 1 representing State Register and 0 representing non-State Register.

2 Pre-Processing

2.1 Features of Designs

Before the method of pre-processing is explained, the features and their meanings will be discussed. A register can be described in more than just the amount of difference between gates and flip-flops that makes it up. To further identify a register, features[9] of the register and its neighbour can be used. In this section, the different types of features will be discussed. Before the discussion, some terminology can be explained as follows.

- Nodes: In a netlist can be represented by a gate or flip-flop with a specified depth. In this thesis, only the nodes represented by flip-flops are of interest.
- Depth of nodes: The number of steps between the last ancestor to be considered and the node.

For each register, a set of structural features were calculated, and these are discussed in the following subsections.

2.1.1 Average Neighbour Degree

Average Neighbour Degree is the average edges of a node in the neighborhood. It is also defined by the following expression.

$$k_{nn,i} = \frac{1}{|N(i)|} \sum_{j \in N(i)} k_j \tag{2.1}$$

Where $N(i)$ are the neighbours of node i and k_j is the degree of node j which belongs to $N(i)$

2.1.2 Betweenness Centrality

Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v. It can be defined by using the expression

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{s,t} \tag{2.2}$$

Where V is the set of nodes, $\sigma(s,t)$ is the number of shortest $(s,t)-paths$, and $\sigma(s,t|v)$ is the number of those paths passing through some node v other than s,t. If $s = t, \sigma(s,t) = 1$, and if $v \in s,t$, $\sigma(s,t|v) = 0$

2.1.3 Closeness Centrality

Closeness centrality 1 of a node u is the reciprocal of the average shortest path distance to u over all $n-1$ reachable nodes.

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)} \tag{2.3}$$

2.1.4 Clustering

For unweighted graphs, the clustering of a node u is the fraction of possible triangles through that node that exist,

$$c_u = \frac{2T(u)}{deg(u)(deg(u)-1)} \tag{2.4}$$

where $T(u)$ is the number of triangles through nodes u and $deg(u)$ is the degree of u.

2.1.5 Degree

The node degree is the number of edges adjacent to the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iterator for (node, degree) as well as lookup for the degree for a single node.

2.1.6 Degree Centrality

Defined as the number of links incident upon a node (i.e., the number of ties that a node has)[**?**]

Let v* be the node with highest degree centrality in G. Let X:=(Y,Z) be the |Y|-node connected graph that maximizes the following quantity (with y* being the node with highest degree centrality in X):

$$H = \sum_{j=1}^{|Y|}[C_D(y*) - C_D(y_j)] \tag{2.5}$$

Correspondingly, the degree centralization of the graph G is as follows:

$$C_D(G) = \frac{\sum_{i=1}^{|V|}[C_D(v*) - C_D(v_i)]}{H} \tag{2.6}$$

The value of H is maximized when the graph X contains one central node to which all other nodes are connected (a star graph), and in this case

$$H = (n-1) \cdot ((n-1)-1) = n^2 - 3n + 2. \tag{2.7}$$

So, for any graph G:=(V,E)

$$C_D(G) = \frac{\sum_{i=1}^{|V|}[C_D(v*) - C_D(v_i)]}{|V|^2 - 3|V| + 2} \tag{2.8}$$

2.1.7 Indegree

The node in-degree is the number of edges pointing in to the node.

2.1.8 Has Feedback Path

A node has a feedback path if there exists a loop from the output of the node to its input.

2.1.9 Katz

Katz centrality computes the centrality for a node based on the centrality of its neighbors. It is a generalization of the eigenvector centrality. The Katz centrality for node i is

$$x_i = \alpha \sum_j A_{ij} x_j + \beta \tag{2.9}$$

where A is the adjacency matrix of graph G with eigenvalues λ.The parameter β controls the initial centrality and

$$\alpha < \frac{1}{\lambda_{max}} \tag{2.10}$$

2.1.10 Load Centrality

The load centrality of a node is the fraction of all shortest paths that pass through that node.

2.1.11 Outdegree

The node out-degree is the number of edges pointing out of the node.

2.1.12 Pagerank

PageRank computes a ranking of the nodes in the graph G based on the structure of the incoming links. It was originally designed as an algorithm to rank web pages.

2.1.13 xx good SN

This feature indicates a good starting node for RELIC/fastRELIC search algorithms. Hence it is not useful in this machine learning use case and therefore will be dropped prior to any feature selections method.

2.1.14 xx state ff

This feature will represent if a register is or is not a state register by assigning 1 or 0 respectively. The goal of the neural network model is to be able to input any arbitrary registers and predict whether the register is or is not a state register, by assigning it a 1 or 0.

2.2 Pre-Processing for Artificial Neural Network

2.2.1 Scaling

Before creating a neural network to train and predict the classification of the registers, a given data set containing the features mentioned in section 2.1 with different number of flip-flops in each file, has to be pre-processed. First the targeted feature in this case "xx state ff" has to be identified and removed from the files followed by scaling down of the data set to values between 0 and 1, using the module "Scikit learn"[6] as the values of some features ranges between 0 to 1 while others between 0 to 10, eg, outdegree has a range between 2 to 5, clustering has a range between 0 to 0.1. Due the difference in ranges for each feature, scaling is necessary to ensure that there is no biasness from the neural network due to one feature having higher values than the other.

2.2.2 Methodology of additional features

5 different implementations of the given data sets listed were chosen to train the neural network.

- Original features present in the data set
- Original features present in the data set with fastRELIC(PSS) as an additional feature
- Original features present in the data set with Cosine Similarity as an additional feature
- Original features present in the data set with Euclidean distance as an additional feature
- Original features present in the data set with Euclidean and fastRELIC as additional features

The additional features are scaled using the same method as the other features. Results of each run will be further discussed in Chapter 4.

Original Features

Previous works have already calculated a set of features for the data set, which can be found in section 2.1, these features will be known as Original features.

fastRELIC

To use the fastRELIC method as an additional feature, the PSS is used. A PSS between 0 to 1 is produced for the pair register and the values produced are influenced by the selected depth of the design in the algorithm. A threshold of 0.7 selected in the experiments and a counting algorithm is used to count the similarity score of a register to other registers above the threshold. These counts are then normalized as not all data sets have the same number of registers. The number of similarity counts can be interpreted by the higher the count, the more registers the targeted register is similar to, therefore, the lower the count, the higher the probability that the register is a State Register.

Cosine Similarity

The cosine similarity is a measurement of cosine angle between 2 non-zero vectors. The result from this measurement determines how similar the 2 vectors are, in the direction they are pointing to. The mathematical expression of cosine similarity can be determined by the manipulation of the inner product mathematical equation.

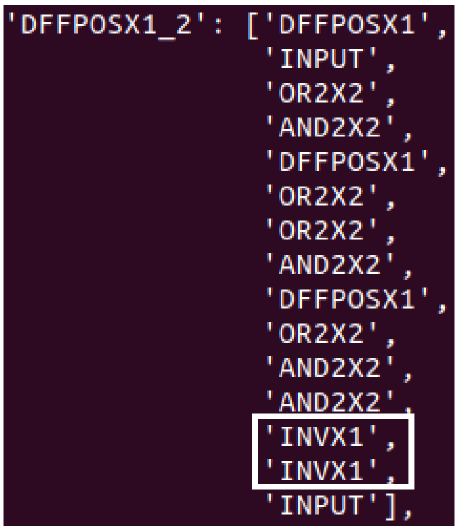
The inner product of 2 n^{th}-dimensional vectors u and v is defined as:

$$u \cdot v = |u||v|cos(\theta) \tag{2.11}$$

Where θ is the angle between u and v, therefore the cosine similarity is given to be:

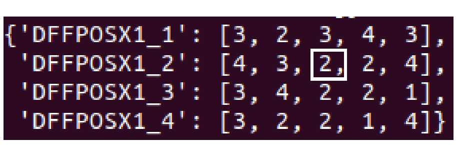
$$cos_{sim}(u, v) = cos(\theta) = \frac{u \cdot v}{|u||v|} \tag{2.12}$$

To use cosine similarity as a feature, first the register's shape shown in Figure 2.1 is extracted by selecting a depth. The register's shape is expressed as a vector with each dimension representing the number of same gates or flipflops the register has, shown in Figure 2.2. In this example, the 2 that is boxed in Figure 2.2 corresponds to the two inverters boxed in Figure 2.1. After calculating the cosine similarity, similar to how PSS is implemented, a threshold is selected, and a counting algorithm is used to count the number of registers each register is similar to. Only a similarity score of more than then the selected threshold will be counted by the counting algorithm. The result is then normalized to ensure fairness.The lower the count, the higher the probability that the register is a State Register.



```
'DFFPOSX1_2': ['DFFPOSX1',
               'INPUT',
               'OR2X2',
               'AND2X2',
               'DFFPOSX1',
               'OR2X2',
               'OR2X2',
               'AND2X2',
               'DFFPOSX1',
               'OR2X2',
               'AND2X2',
               'AND2X2',
               'INVX1',
               'INVX1',
               'INPUT'],
```

Figure 2.1 Register shape



```
{'DFFPOSX1_1': [3, 2, 3, 4, 3],
 'DFFPOSX1_2': [4, 3, 2, 2, 4],
 'DFFPOSX1_3': [3, 4, 2, 2, 1],
 'DFFPOSX1_4': [3, 2, 2, 1, 4]}
```

Figure 2.2 vector

However, this method is not implemented in the final testing. As from initial testing, the count value for the method of cosine distance does not differ significantly between any two registers, only by $< 10\%$. This resulted in a similar count score, for all registers, depending on the selected threshold, which was selected to be 0.9. This led to poorer performance at times when compared to the original files initial testings. Refer to Section 4.1.1 for testing methodology.

Euclidean Distance

The euclidean distance of 2 vectors is the measurement of the shortest distance between 2 points of the vector. The closer they are, the larger the similarity. In Figure 2.3, the euclidean distance between 2, 2-dimensional vectors, u and v is represented by d.

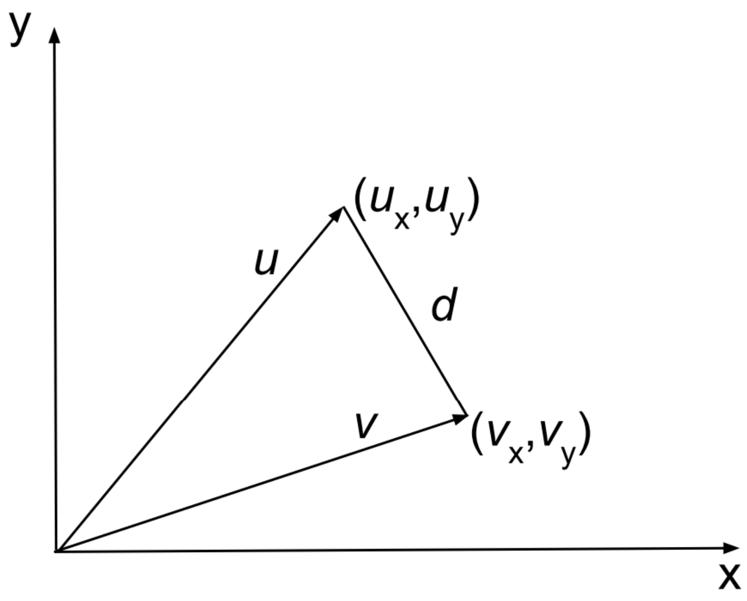


Figure 2.3 Euclidean Distance

The distance d can be calculated by

$$E(u,v) = \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2} \tag{2.13}$$

In a n^{th}-dimensional vector, it can be represented by

$$E(u,v) = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2} \tag{2.14}$$

Following the extraction method for the register shape, which is then expressed as a vector detailed in Section 2.2.2, the euclidean distance between the any 2 register's vector can be calculated using Equation 2.14. A threshold of 2.4 is then selected for the experiments and a counting algorithm is used. The count is incremented if the euclidean distance is below the selected threshold, as this implies the compared vectors are close and thus similar. The lower the count, the higher the probability that the register is a State Register.

2.2.3 Feature Selection

Feature selection has been shown to be one of the most important steps before implementing the model as it will severely impact the training of the model. The benefits of doing feature selection is listed as follows[7].

- Reduces Overfitting: Less redundant data, lesser the opportunity for the network to be influenced by these features
- Improves Accuracy: Less misleading data improves model accuracy as the neural network does not have to take into account the patterns of these features
- Reduces Training Time: Less data points reduce algorithm complexity resulting in reduced time taken to train the model

Due to the clear advantages of performing feature selection, five methods of feature selection were performed.

- Constant Filtering
- Quasi-Constants Filtering
- Feature Permutation
- Sequential Feature Selection

Constants Filtering

Constants Filtering was first decided as is computationally inexpensive yet provides a simple and fast approach. This method was based on the concept that constant values in a feature, does not add any value to the neural network as it does not present a pattern to the neural network. Hence removing such features will reduce the training time of the neural network. This was done via the module "scikit-learn" for python.

Quasi-Constants Filtering

Based on similar reasoning in using constants filtering, quasi-constant filtering method differs from constant filtering by adding a threshold. If the difference between any 2 features is not more than a selected threshold in percentage, the feature will be removed.

Feature Permutation

Feature Permutation is based on the concept whereby permuting the values in a feature breaks the relations between the permuted feature and the rest of the features. This loss of relation should decrease the accuracy significantly (more than a selected threshold) if the feature is important and increase/remain constant if the feature has no impact or hinders the neural network. To determine if a feature is important, the two methods of testing the feature importance is detailed in Section 4.2.3. Each feature in the data set is permuted individually, fitted, and tested on a neural network for k runs. The average accuracy after k runs for each feature is recorded and compared individually to the unpermuted feature set average accuracy after k runs.The feature will be removed if the accuracy is the same, or when the difference is below the selected threshold. The flow of the process can be seen in Figure 2.4.

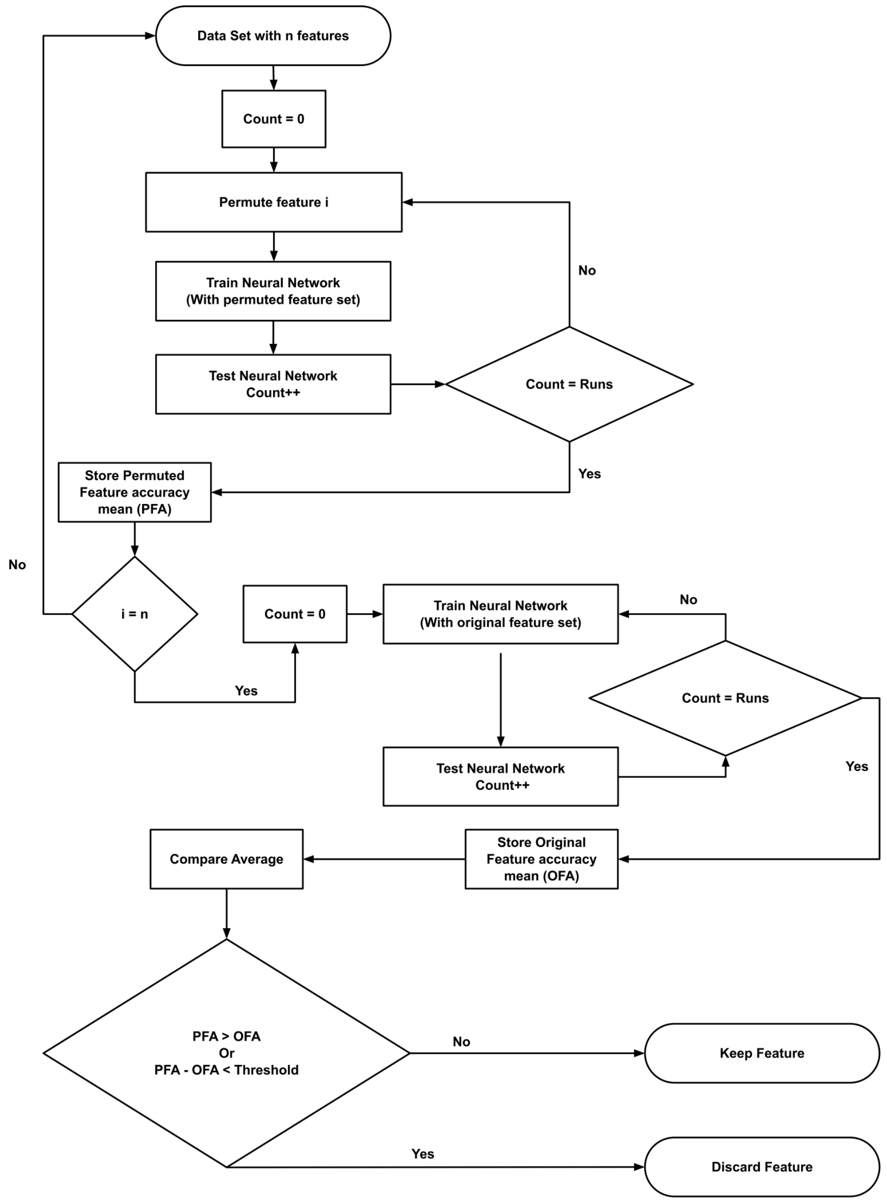

Data Set with n features
Count = 0
Permute feature i
Train Neural Network (With permuted feature set)
Test Neural Network Count++
Count = Runs
No
Yes
Store Permuted Feature accuracy mean (PFA)
No
i = n
Yes
Count = 0
Train Neural Network (With original feature set)
No
Count = Runs
Yes
Test Neural Network Count++
Compare Average
Store Original Feature accuracy mean (OFA)
PFA > OFA Or PFA - OFA < Threshold
No
Keep Feature
Yes
Discard Feature

Figure 2.4 Feature Permutation Flow Chart

In Figure 2.4, the data set is fed to the neural network and a counter is initialized to 0. The first of the n-features is permuted and trained on the neural network. An accuracy is produced by testing the neural network and this process is repeated for a selected number of runs (runs = 100 for the experiments). Once the counter equals to the number of selected runs, an average accuracy will be calculated and the next feature is selected. This process is repeated for all features. Next the unpermuted data set is trained and tested on the neural network for the same number of runs and an average accuracy is calculated. The average permuted accuracy of each feature is compared to the unpermuted average accuracy and the feature will then be discarded or retained based on the set of conditions.

Sequential Feature Selection

Sequential Feature Selection selects the best features combination of feature set, based on a recursive combinations testing. To implement Sequential Feature Selection, a module "mlxtend"[8] was chosen. First the algorithm evaluates the neural network performance on each feature and the highest performing feature is selected. Next the selected feature will be in a combination two with all other features and the performance of the neural network is evaluated again. This process will then be carried out until the desired number of features is reached.

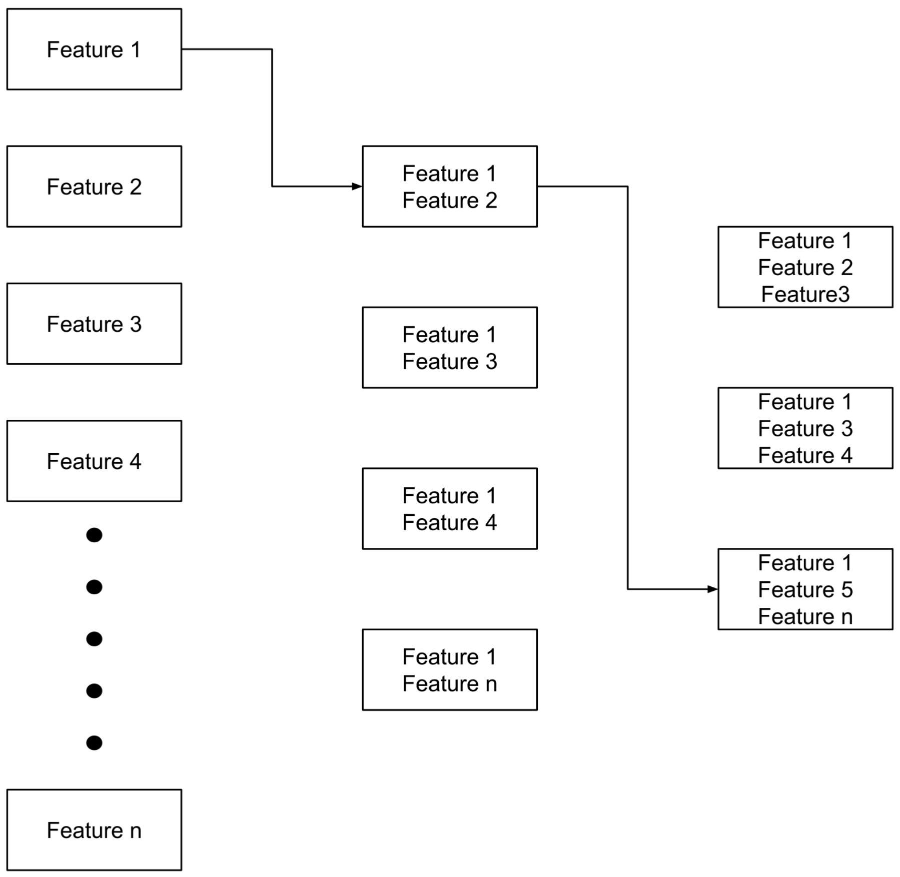


Figure 2.5 Sequential Feature Selection Flow Chart

In Figure 2.5, Sequential Feature Selection is performed on a data set with n-features. Evaluation is done individually for all features and the feature with the highest accuracy is chosen, which is feature 1 in Figure 2.5. A combination of two, containing feature 1 and one other feature is evaluated again, selecting the combination with the highest accuracy. This process is continued until the selected amount of feature provided by the user is reached. In the experiments for this thesis, the upper limit of the number of features is selected to be all features after the filtering method. Based on the results in Section 4.2.4, the average number of features selected from the data set lies between 8 to all features after implementing this feature selection method.

3 Neural Network Model

3.1 Understanding the Neural Network

Machine Learning[**?**] uses data to train a model which can be then used to identify patterns and predict a future set of values. Machine learning can generally be classified into 2 types, supervised and unsupervised. In supervised machine learning problems, the problems can then be further classified into 2 different types of problems, regression, and classification. In regression machine learning problems, the model's predicted outcomes using the trained data are typically real numbers, while classification machine learning data outcomes are usually classes or categories of which the data belongs to. In unsupervised machine learning, a data set is passed and by applying different algorithms and statistical data, the goal of an unsupervised machine learning model is to get the model to identify the structure of the given data set.

The model used to classify if a register is or is not a state register will be a supervised machine learning model. Machine learning is generally only a single layer, using an estimator (classification or regression) to perform predictions and requires humans to input the best feature to learn from. Deep learning, a subset of machine learning on the other hand uses multiple layers by connecting neurons between layers, interacting directly with the data set to learn what are the most important features. Deep learning uses the neural network to perform prediction and an optimizer to back propagate to adjust the weights and bias for better predictions. Figure 3.1 shows a single neuron neural network with multiple different hyperparameters such as activation function, threshold and transfer functions, these hyperparameters will be further explained in details in Section 3.2. By adding more layers and neurons, the neural network is then known as a multilayer perceptron neural network. The general diagram for a multilayer perceptron neural network can be seen in Figure 3.2.

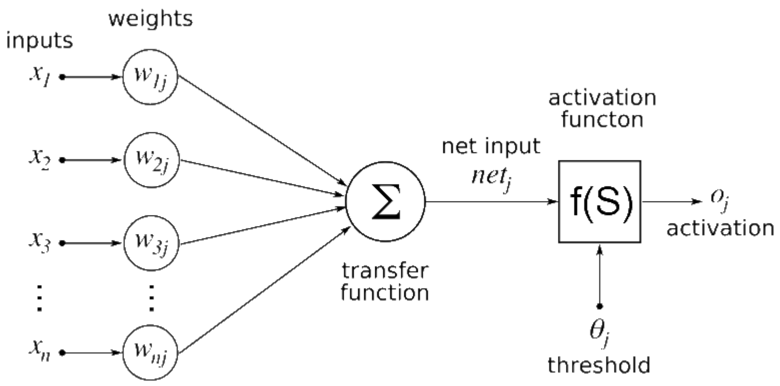


Figure 3.1 Single Neuron Neural Network[10]

The mathematical expression of a single neuron neural network is given to be:

$$f(S) = f(b + \sum_{i=1}^{n} x_i w_i) \tag{3.1}$$

which is based on the equation of a straight line.

There are multiple different architectures for a neural network, such as Deep Feed Forward Neural Network (FNN), Convolution Neural Network(CNN), Recurrent Neural Networks(RNN) and many others, each having their own unique designs, suited for different purposes. Due to the nature of this thesis, of using Directed Graphs, the architecture of FNN is selected. To create this neural network, layers of densely connected neurons are stacked together. This is depicted in Figure 3.2

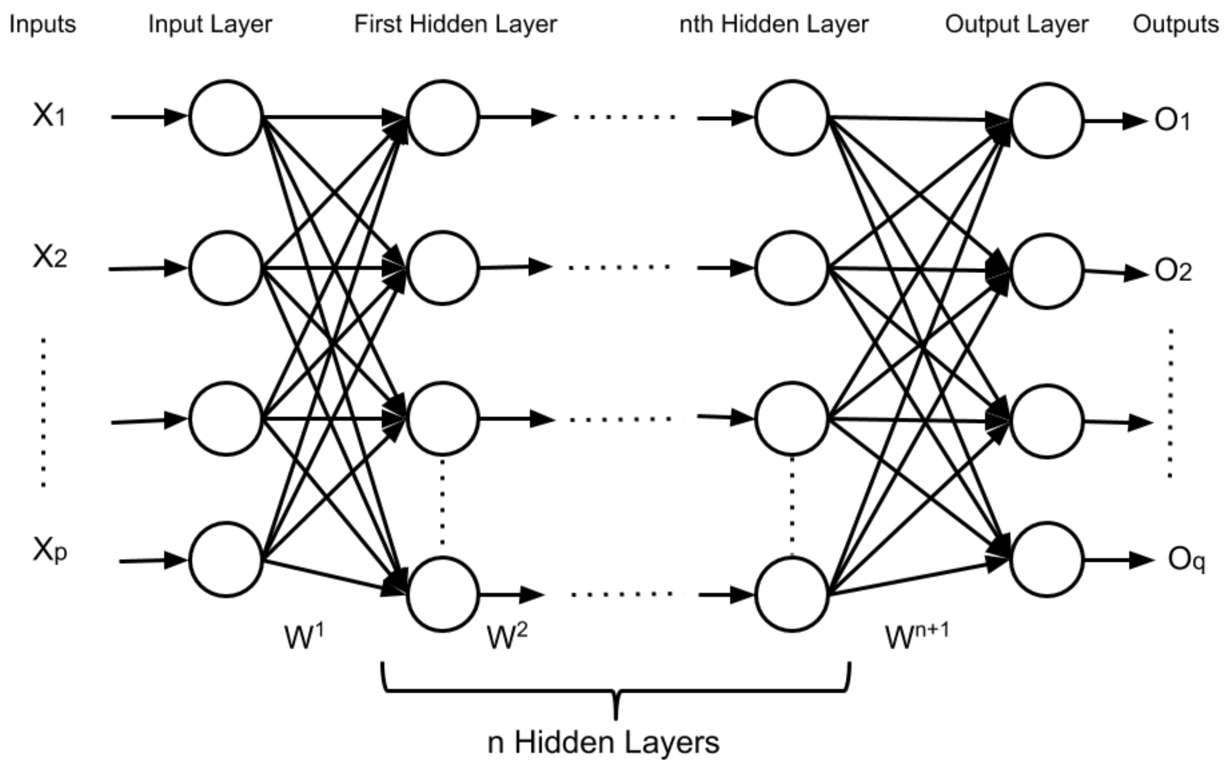


Figure 3.2 Deep Feed Forward Neural Network[11]

The question of how many layers there should be or what is the most suitable layer for the data set the neural network is training on arises. The most suitable amount of layer cannot be stated as easily as it depends on the complexity of the data set. In general, the more layers there are, the higher the abstract representation of the input the neural network can build. However, this does not mean that the more the layers, the better the trained model will be. If a data set is of low complexity and is fed through a neural network of high layers, there is a very high chance of overfitting the model. Thus, this is a parameter to consider when training the neural network.

3.2 Hyperparameters

Aside from deciding on the number of layers in the neural network, there are multiple other parameters known as hyperparameters in a neural network to tune. In this section the various hyperparameters and how they affect the neural network will be discussed.

3.2.1 Activation function

As shown in Figure 3.1, there is a function $f(s)$ represented as $f(x)$ in the following explanation of the different types of activation functions. The role of the activation function is to map the output of a single neuron to a more suited value, adding non-linearity to the model. There are multiple different activation functions, however only the most used activation function will be discussed.

Types of activation function:

- Sigmod
- ReLU
- ELU
- tanh

Sigmod

The sigmoid function maps the output of the neuron to a value between 0 and 1. This function is particularly useful in a binary situation, e.g. (true or false), (yes or no), (1 or 0). However the function does have a vanish gradient problem as shown in the derivative of the function in Figure 3.3.

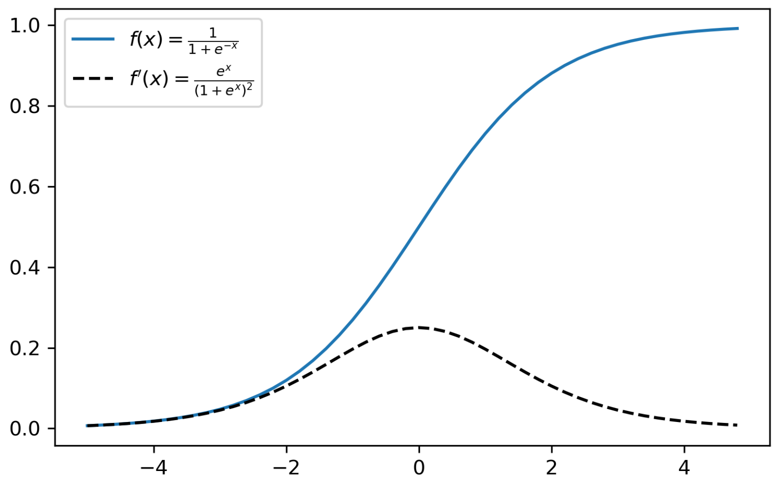


Figure 3.3 Sigmoid Graph

Vanishing gradient problems occur when more layers using certain activation functions are added. This will then cause the gradient of loss function to approach 0, making the training of the neural network harder.

ReLU

ReLU is the most common activation function in a neural network as it returns the input value when the input is more than 0 and 0 if the input is less than 0. ReLU can be used especially when the designer of the neural network is unsure of what activation function to use. ReLU also comes with its benefits of being non-computationally intensive and due to its linearity, does not have the vanishing gradient problem as shown in Figure 3.4.

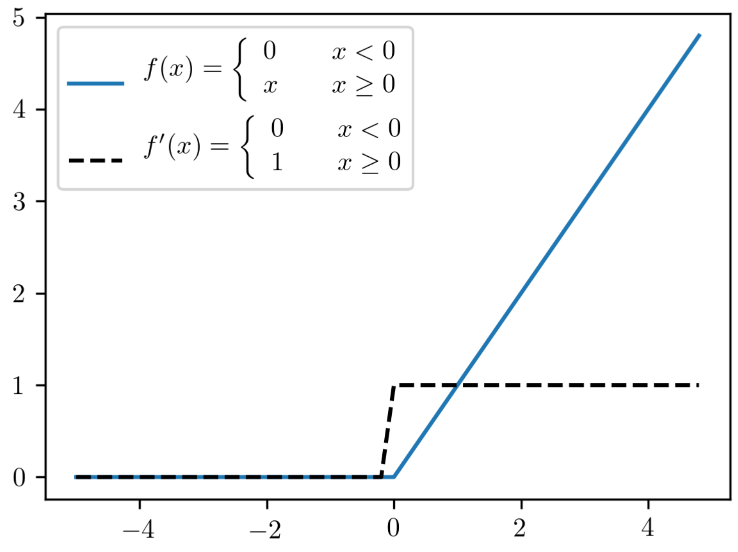


Figure 3.4 ReLU Graph

ELU

ELU is very similar to ReLU, hence can be a good alternative to ReLU. The difference between both is that ELU can be scaled and produces negative output for negative values of the input. However, the negative output does saturate given a large enough negative input. The function and its derivative can be represented by the following mathematical equation and Figure 3.5, where $\alpha = 1$.

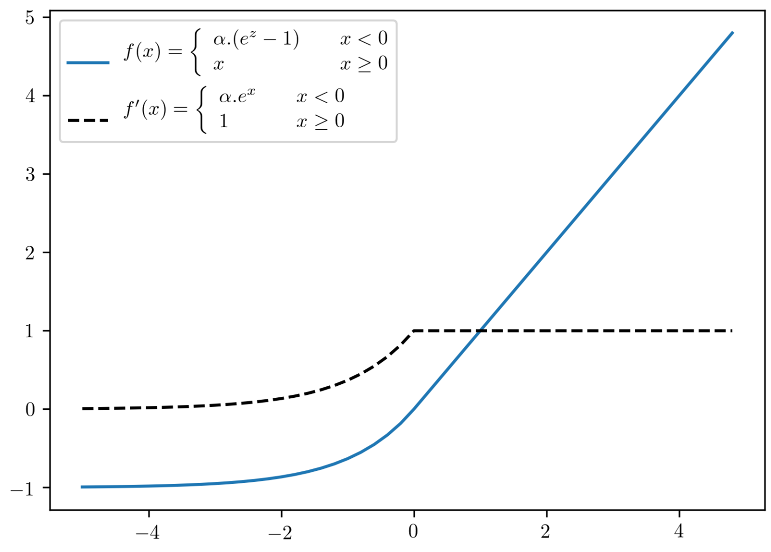


Figure 3.5 ELU Graph

Hyperbolic Tangent

The output of the hyperbolic tangent function is similar to the sigmoid function, such that the output is mapped between 2 values, -1 and 1. As compared to the sigmoid function, the hyperbolic tangent function has a steeper gradient. The function and its derivative can be represented by the following mathematical equation and Figure 3.6.

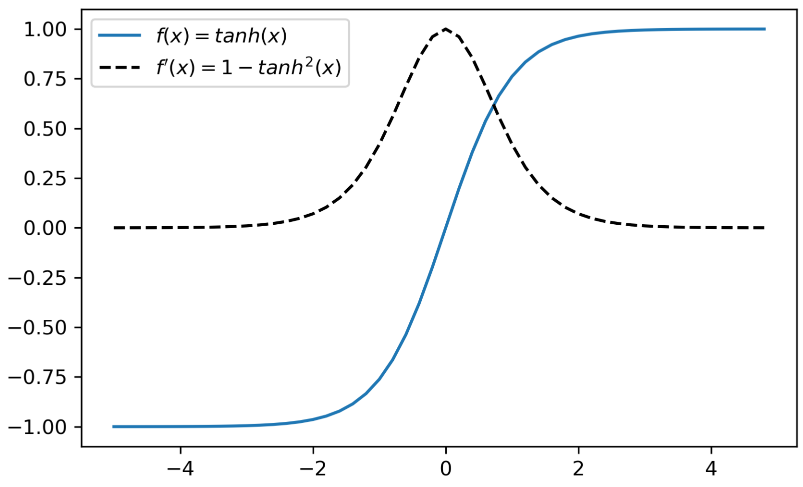


Figure 3.6 tanh(x) Graph

3.2.2 Loss Function

The lost function is a measurement hyperparameter, to determine the amount of error between the predicted output and the actual output of the neural network. Choosing the correct loss function, can affect how well the model is viewed.

Some common examples of the loss function are:

- Binary Cross Entropy
- Mean Square Error
- Sparse Categorical Cross Entropy

The selection of which function to use as a measurement is dependent the what the problem is.

3.2.3 Optimizer

Since the loss function is used to determine the capability of the neural network, the question of can the loss be reduced arises. This is where the optimizer helps. The optimizer role is to find the global minimum of the loss function. This is done by using the chain rule to back propagate, modifying the weights of the neural network.

Some common examples of the optimizer are:

- Adam
- RMSProp
- Stochastic Gradient Descent

3.2.4 Epoch

Epochs represent the amount of times the neural networks see the training data. Depending on the complexity of the data set, a high number of epochs might result in overfitting. This means the trained model will do particularly well in evaluating the trained data set, but once it is evaluated using the testing data, the model performs poorly. This can be better illustrated in a Figure 3.7 for under-fitting, optimal-fitting and over-fitting.

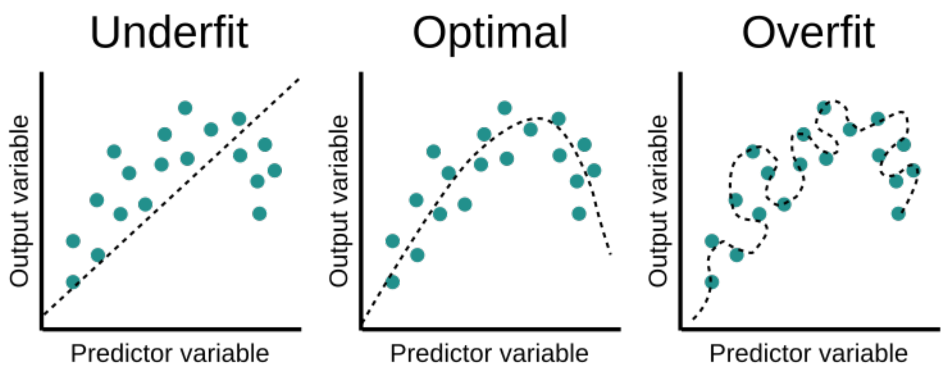


Figure 3.7 Fitting

3.2.5 Batch Size

Batch size is the number of samples that are passed to the network at once. Generally, the larger the batch size, the faster the neural network completes each epoch during training. However, if the batch size is too large, the quality of the neural network may degrade, as it is unable to generalize on the data it has not seen before.

3.2.6 Number of Hidden Layers

The number of hidden layers refer to the number of layers between the input and output layers.

3.2.7 Number of Neurons

The number of neurons per layer can differ and should be optimized depending on the usage of the neural network.

3.2.8 Dropout

Dropout is a regularization technique to prevent over fitting. Dropout removes some neurons in the layers to prevent the over-reliant on any neuron.

3.3 Overview of Neural Network

With the knowledge of the neural network and its hyperparameters, the flow of how to train and validate a neural network can be discussed. After the pre-processing of the data, the data has to be split into 2 groups, the training set and validation set.

The training and validation data can be defined as follows:

- Training data: Data from a data set which the neural network sees and uses for pattern recognition.
- Validation data: Data from a data set which is not seen by the network, which is used to test the accuracy of train model to provide further insides of the neural network, so that some hyperparameter tuning can be done

To train a neural network based on the data set, the recommended practice is to train the model with 80% of the data set and test with 20% shown as pre-processing in Figure 3.8. This will allow for cross-validation if needed. Once the data is split into groups, it can then be used to train the neural network as shown. A model will be deployed as a trained model and can be validated with the validation set. If the results are not satisfactory, the hyperparameters have to be tuned and the model will have to be retained. In Figure 3.8, shows the overview of how the neural network is trained and validated.

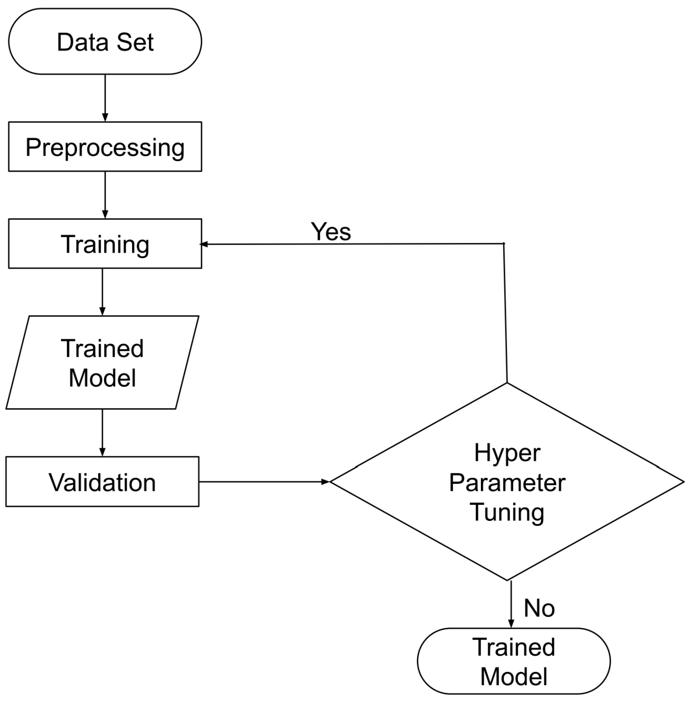


Figure 3.8 Neural Network Flowchart

However, due to the limited set of data and the structure of the files, the method of splitting data into 80% for training and 20% for testing, was not adopted. The method of training and testing adopted will be further discussed in Chapter 4

4 Results

In Chapter 2, the pre-processing methodology was discussed. In this chapter, the process and results will be discussed.

4.1 Data Manipulation

As mentioned in Chapter 2, Section 2.1, the given data consist of the listed feature and each feature holds a certain value for each register in the data file. There is a total of 13 data files given and 12 were used to train, while 1 was used to test. Prior to training, the python module "pandas”[12] was used to implement the additional feature methods in Section 2.2.2. The data in 12 training files is then collated into a single collective file. The feature "xx state ff" was subsequently removed from the collated training file and the test file and stored as the target for the neural network. The feature "xx good SN" was dropped as mentioned in Chapter 2, as it is not a feature the neural network should rely on. The data were then scaled using the scaling process.

4.1.1 Implementing Additional Features

To test which of the additional features implementation performs the best, a neural network was set up using a python module "Keras"[13] with the hyperparameters shown in 4.1.

Hyperparameter	Input
Optimizer	RMSprop
Loss Function	Binary Crossentropy
Batch Size	12
Epochs	22
Dropout	0
First Neurons	66
Hidden Layers	12
Hidden Neurons	50
Hidden Layer Activation Function	ReLU
Last Activation Function	Sigmoid

Table 4.1 Neural Network Hyperparameters

The performance of the implementations can be determined by the accuracy scores of the neural network, given by the mathematical expression:

$$\text{Model Accuracy} = \frac{\text{Number of Correctly Predicted Registers}}{\text{Total Number of Registers}} \tag{4.1}$$

Since this thesis is about identifying state registers, it is interesting to observe the percentage of state accuracy out of all state registers the neural network correctly predicted as well. The mathematical expression is given to be:

$$\text{State Register Accuracy} = \frac{\text{Number of Correctly Predicted State Registers}}{\text{Total Number of State Registers}} \tag{4.2}$$

The hyperparameters shown in Table 4.1 were chosen based on the previous experimental runs of using the hyperparameter tuning module "Talos"[14]. Previous experimental runs' hyerparamters tested are shown in Table 4.2, refer to Section 4.3 for detailed explanation of "Talos".

Hyperparameter	Input
Optimizer	Adam, RMSprop
Loss Function	Binary Crossentropy, Sparse Categorical Crossentropy
Batch Size	12, 24,32
Epochs	10 to 26, increment of 4 per step
Dropout	0, 0.25, 0.5
First Neurons	10 to 66, increment of 4 per step
Hidden Layers	1 to 16, increment of 4 per step
Hidden Neurons	10, 20, 30, 40, 50
Hidden Layer Activation Function	ELU, ReLU
Last Activation Function	Sigmoid

Table 4.2 Talos Parameters

To test which implementation provides the best results, a consistent set of features across all implementations has to be ensured. Hence, the testing of the different implementations was done prior to feature permutation method and sequential feature selection method, but after constant filtering method.

12 files for training and 1 file for testing were rotated, with each file training and testing the model 100 times, e.g, files B–N were used to train the neural network while file A was used to test the neural network, followed by files A, C–N were used to train the neural network while file B was used to test the neural network.

The results were stored and the model in "Keras" backend was cleared after every training and testing to ensure previous training and test results does not affect the current run. The average accuracy scores of 100 runs per file were collated and the process was repeated 5 times to ensure consistency. The results of all implementations with the average accuracy after 5×100 runs is shown in Tables 4.3 and 4.4.

From Table 4.3, it can be observed that the average model accuracy increased substantially up to 12% with either of the other 3 additional features methods of implementation when compared to the original implementation. Taking a benchmark of 70% as "great performance", the implementation "with fastRELIC" and "with Euclidean and fastRELIC", shows only one file performing below that benchmark, while the implementation of "with Euclidean" has 3 files. Comparing the results, it shows a substantial improvement over the original implementation.

Using the benchmark of 70% for the state registers accuracy as a "good performance" reference, from Table 4.4 the implementation "with fastRELIC", though has scored a lower average accuracy across all files by 1% compared to "with Euclidean and fastRELIC", it has a total of nine files, one more file that scored above the 70% benchmark. Overall, all three implementations show a substantial improvement in the number of correctly predicted state registers when compared to the original implementation.

File Name	Model Accuracy			
	Original	With fastRELIC	With Euclidean	With Euclidean and fastRELIC
b01_reset.csv	0.95	0.97	0.98	0.97
b02_reset.csv	0.95	0.96	0.93	0.92
b04_reset.csv	0.56	0.80	0.98	0.99
b06_reset.csv	1.00	1.00	1.00	1.00
b08_reset.csv	0.59	0.90	0.84	0.91
b09_reset.csv	0.57	0.72	0.62	0.73
b10_reset.csv	0.73	0.79	0.76	0.83
b14_reset.csv	0.57	0.67	0.61	0.67
completogpio.csv	0.84	0.89	0.86	0.87
FSM.csv	0.64	0.80	0.72	0.84
MEMORY_INTERFACE.csv	0.93	0.96	0.96	0.96
spi_axi_master.csv	0.98	0.97	0.96	0.92
uart.csv	0.45	0.72	0.56	0.70
average	0.75	0.86	0.83	0.87

Table 4.3 Model Accuracy

File Name	State Register Accuracy			
	Original	With fastRELIC	With Euclidean	With Euclidean and fastRELIC
b01_reset.csv	0.92	0.95	0.96	0.95
b02_reset.csv	0.94	0.95	0.90	0.90
b04_reset.csv	0.38	0.91	0.56	0.82
b06_reset.csv	0.99	1.00	0.99	1.00
b08_reset.csv	0.50	0.60	0.33	0.62
b09_reset.csv	0.84	0.92	0.98	0.97
b10_reset.csv	0.88	0.86	0.93	0.87
b14_reset.csv	0.47	0.21	0.25	0.18
completogpio.csv	0.49	0.99	1.00	1.00
FSM.csv	0.38	0.76	0.55	0.84
MEMORY_INTERFACE.csv	0.29	0.58	0.53	0.61
spi_axi_master.csv	0.33	0.52	0.73	0.67
uart.csv	0.25	0.74	0.46	0.68
average	0.59	0.77	0.71	0.78

Table 4.4 State Register Accuracy

4.2 Feature Selection

To attempt to improve the results obtained in Table 4.3 and 4.4 and reduce the training time, different methods of feature filtering were implemented.

4.2.1 Constant Filtering

Using the constant filtering method, on the data sets, the constant feature "Degree" was removed from both training and test data frame.

4.2.2 Quasi-Constant Filtering

Using the quasi-constant filtering method, with a variance threshold of only 1% more than half of the feature sets were removed. This gave the neural network for feature permutation a very limited set of features to work with, resulting in a very poor performance in initial runs. Hence, the use of this filtering method was not implemented.

4.2.3 Feature Permutation

To implement feature permutation ensuring consistency, all 13 files were rotated with 12 files training and 1 file testing, repeating for a total of 5 times per file. All training files have their features individually permuted for a 100 times, followed by training the neural network and tested for accuracy with the testing set per permutation. The average accuracy of these 100 permutations per feature is then compared with the original data set, which underwent training and testing the same model for a 100 times for 5 times. The algorithm will then determine if the feature is a hindrance or not based on the explained methodology in section 2.2.3. Refer to Figure 2.4 for the flowchart.

From the four implemented methods for additional features (including original), two methods of calculating the importance of a feature was used.

Determining Feature Importance Method 1

The first method can be expressed in ratio, using the following equations:

$$R_n(A_{original}, A_{permuted}) = \frac{A_{original}}{A_{permuted}} \tag{4.3}$$

$$SRR_n(SRA_{original}, SRA_{permuted}) = \frac{SRA_{original}}{SRA_{permuted}} \tag{4.4}$$

Where n is the feature, R_n is the ratio between the Model Accuracy(4.1) of the original feature and permuted features, SRR_n is the ratio between the State Register Accuracy(4.2) of the original features and permuted features.

The ratio for both equations, can then be interpreted by:

$R_n < 1$	$SRR_n < 1$	Feature Hindrance
$R_n = 1$	$SRR_n = 1$	Feature Hindrance
$R_n > 1$	$SRR_n > 1$	Feature Important

Table 4.5 Ratio Interpretation

The ratios was calculated by averaging across all the files and runs per feature per implementation method, which is illustrated in Figure 4.1 and Figure 4.2.

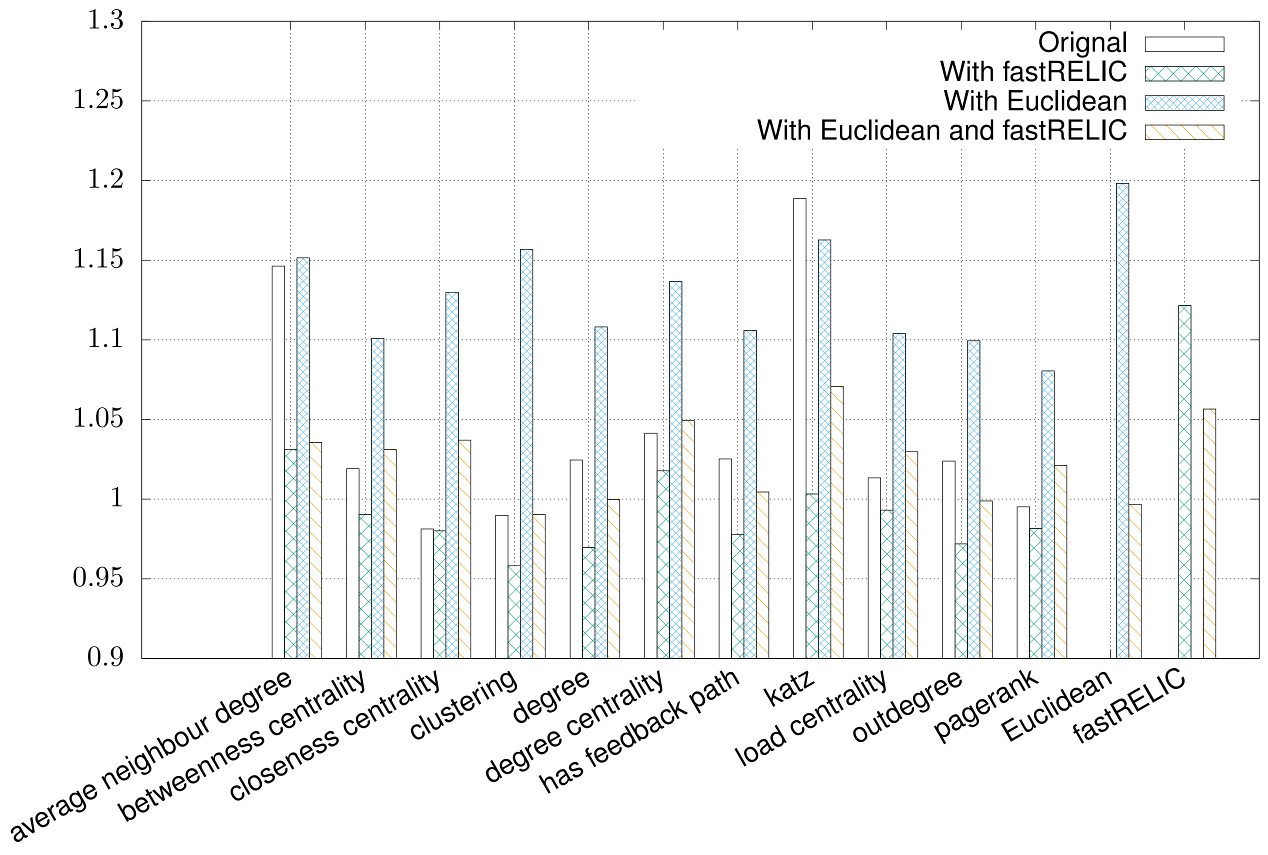


Figure 4.1 Model Accuracy Ratio (R_n) Per Feature

In Figure 4.1, focusing on the primary(non-additional) features, it shows general inconsistency across the different implementations, with different implementation viewing different features as important, referencing Table 4.5. Additionally, with the additional feature(s) implementations, the importance of other features were significantly affected as illustrated. If features below 1 are removed from the respective implementations, this will drastically affect the neural network for the respective implementation in a negative way. Hence, further exploration is needed due to the lack of a clear trend of which features are of lesser importance.

Though there is a general inconsistency in feature importance as shown in Figure 4.1, the 3 features which are above the value of 1 across all implementations are "average neighbour degree", "degree centrality" and "katz". Therefore, a conclusion of these 3 features being high importance can be drawn based on R_n.

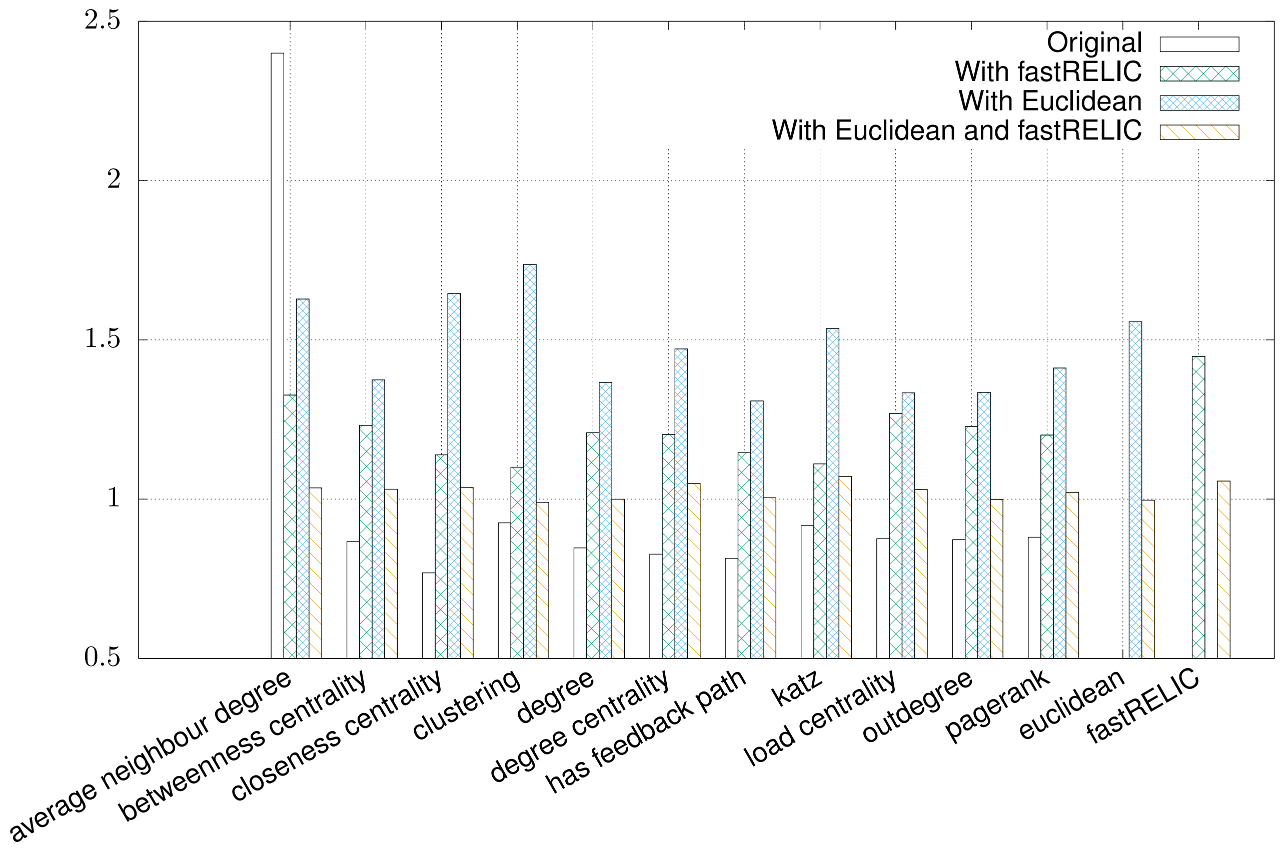


Figure 4.2 Model Accuracy: State Register Ratio (SRR_n) Per Feature

Applying method 1 on the State Register Accuracy instead of the Model Accuracy yields Figure 4.2, which upon first observation seem to be a very different graph, compared to Figure 4.1. This significant value difference can be explained by the amount of state registers in a file. Given that the number of state registers in a file is smaller than the total of all registers type in a file, a value change will cause a significant difference in the final ratio. Therefore, instead of solely focusing on the values, the trend should be taken into consideration as well. Comparing Figure 4.2 to Figure 4.1, despite the files "with Euclidean" having significantly higher ratio in Figure 4.2 compared to Figure 4.1, the ratio values for each feature remains more than 1, which is similar in both figures. For the file "with fastRELIC", all feature ratio is now above 1 thus, a definite conclusion of those features below the value of 1 in 4.1 being less important cannot be drawn.

Determining Feature Importance Method 2

The second method mentioned determines the percentage of feature occurrence across the files to assess feature importance. A counting algorithm is used to count the total amount of times the feature appears after feature permutation selection based on a set of conditions explained in Section 2.2.3, shown in Table 4.6.

The conditions are as follows:

$A_{original} < A_{permuted}$	Feature Hindrance
$A_{original} = A_{permuted}$	Feature Hindrance
$A_{original} > A_{permuted}$, with a difference of $> 1\%$	Feature Important

Table 4.6 Conditions for filtering

The percentage of occurrence can be expressed by:

$$C(n) = (\frac{1}{k}\sum_{i=0}^{k}[f_i = n]) \tag{4.5}$$

where n is the feature, k is the total amount of runs of all files, thus $k = 65$ in this experiment.

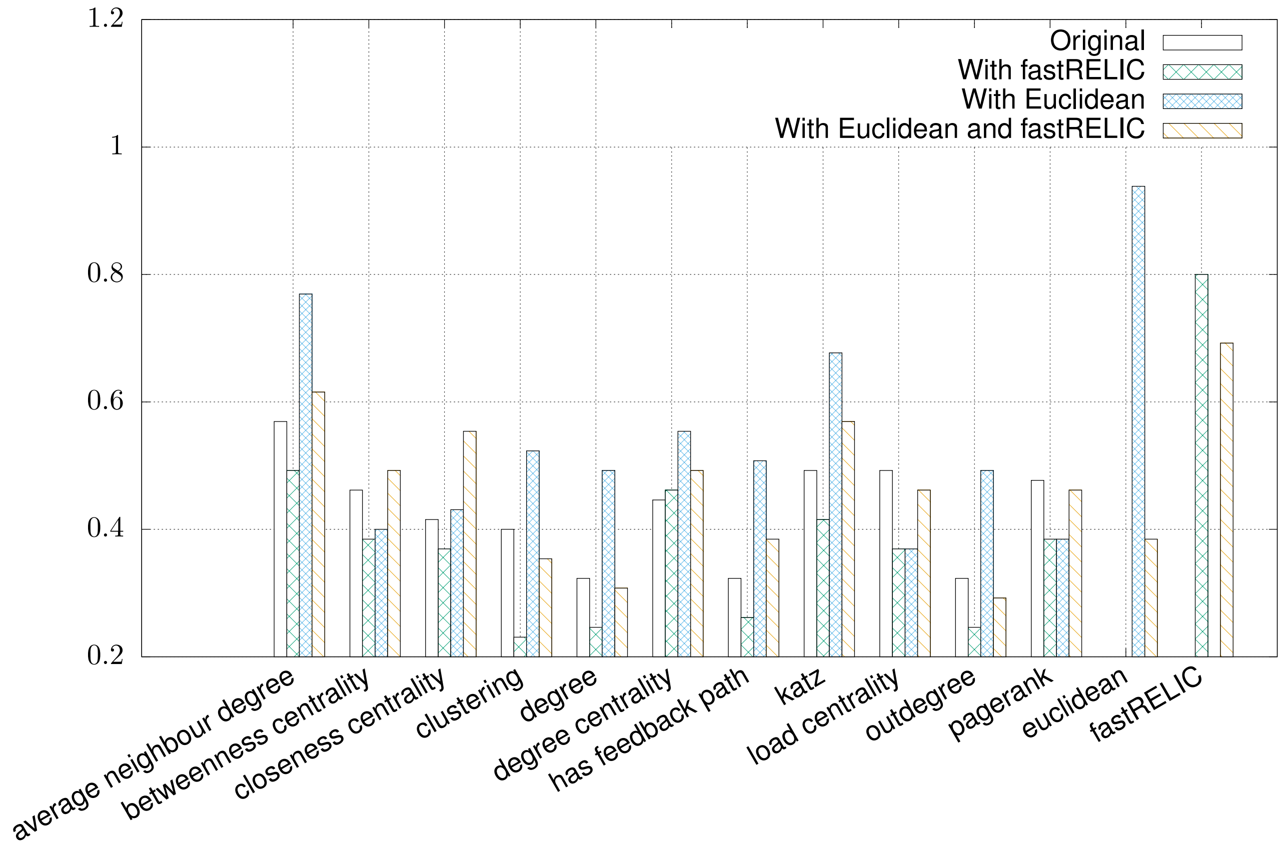


Figure 4.3 Feature Occurrence Percentage: FP Model

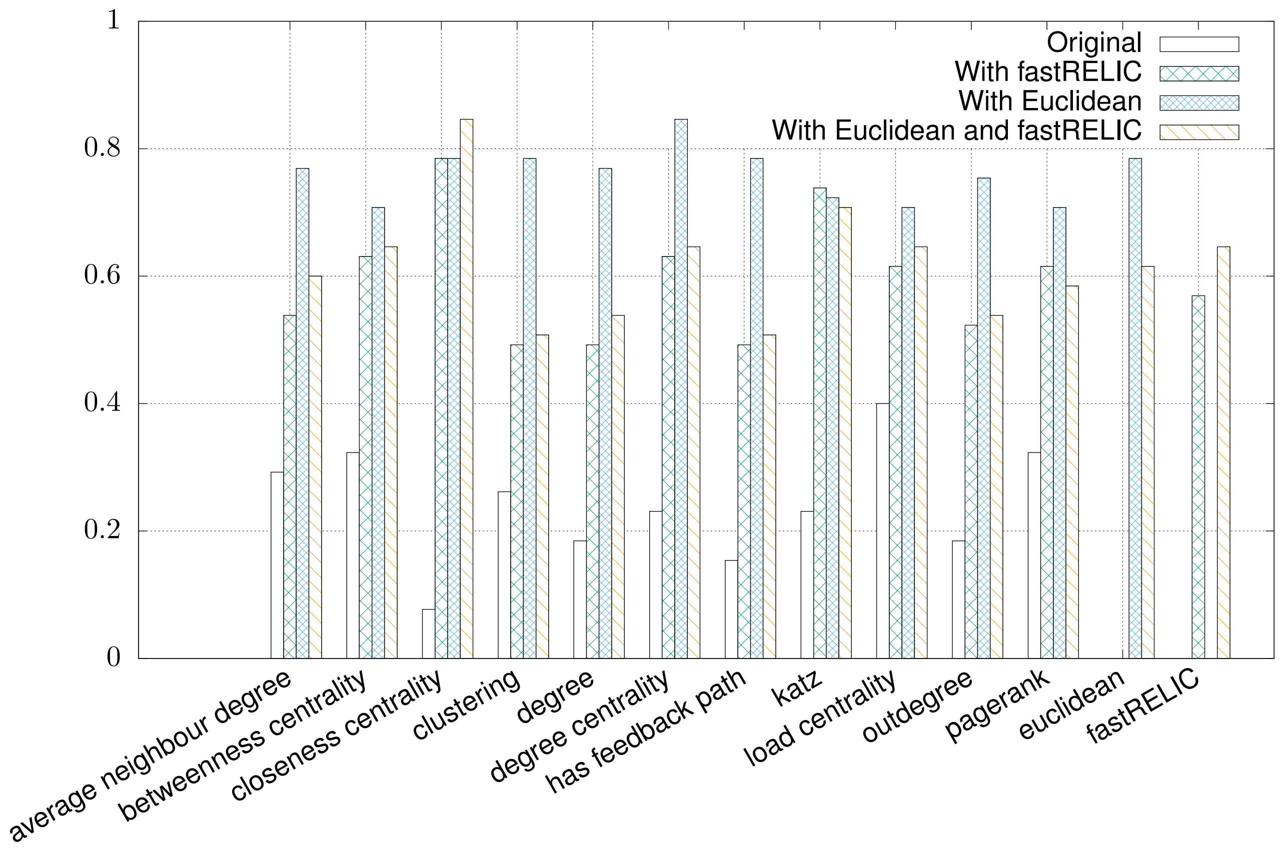


Figure 4.4 Feature Occurrence Percentage: FP State Register

From Figure 4.3 , similar to determining the feature importance based on the ratio shown in Figure 4.1, there is no clear trend of which features are the least important across all the implementations. However, the results shown do support the observations in section 4.2.3,that the features "average neighbour degree" and "katz" are 2 of the more important features across all implementations. It is also worth noting that the feature "degree centrality" might be a feature that carries high importance as well.

In Figure 4.4 the files "With fastRELIC" and "With Euclidean", has view most features to be important when predicting state registers, resulting in about 50% or more for all features. These results are aligned with the results shown in 4.2 where all features are viewed to be important.

Conclusion for Results from Feature Permutation

Using feature permutation method is a way to break the correlations between features, allowing to find the most and least important features, reducing the feature set. However, from this experiment, there is a discrepancy of the features importance when comparing the data between model accuracy and state register accuracy methods. Hence a clear conclusion of which features is of least importance cannot be drawn. Further exploration is required to select features using Feature Permutation methodology.

4.2.4 Sequential Feature Selection

Sequential Feature Selection was initially implemented after feature permutation, to further reduce the feature set and find the best combination of features. However, this resulted in less than low occurrence for almost all features. To illustrate this, method 2 of determining feature importance is used.

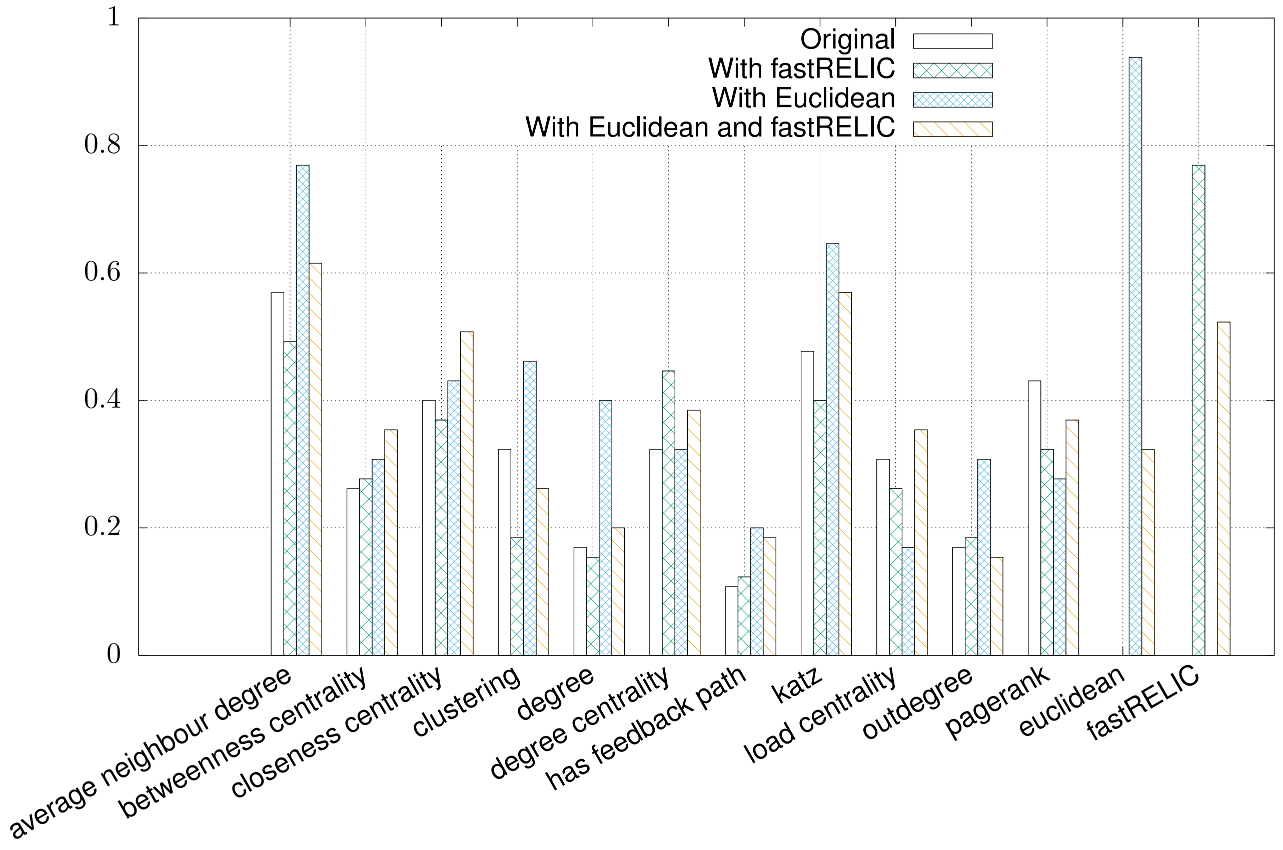


Figure 4.5 Feature Occurrence Percentage: Sequential Feature Selection after Feature Permutation

As observed in Figure 4.5, all but two of the primary features appears more than 50% of across all implementations. Though the results support the importance of the feature "average neighbour degree" and " and katz". A test with fewer than 7 features has shown to be poor performing in the pre-optimized neural network with less than 20% accuracy. Thus, SFS is used independently to support the findings of feature permutation.

To apply Step Features Selection independently, ensuring consistency, Sequential Feature Selection was repeated 5 times per file. Method 2, feature occurrence percentage is then applied to determine overall feature importance. Using the same mathematical equation in Equation 4.5, the percentage of occurrence for all features across all files and runs were calculated and is presented in Figure 4.6.

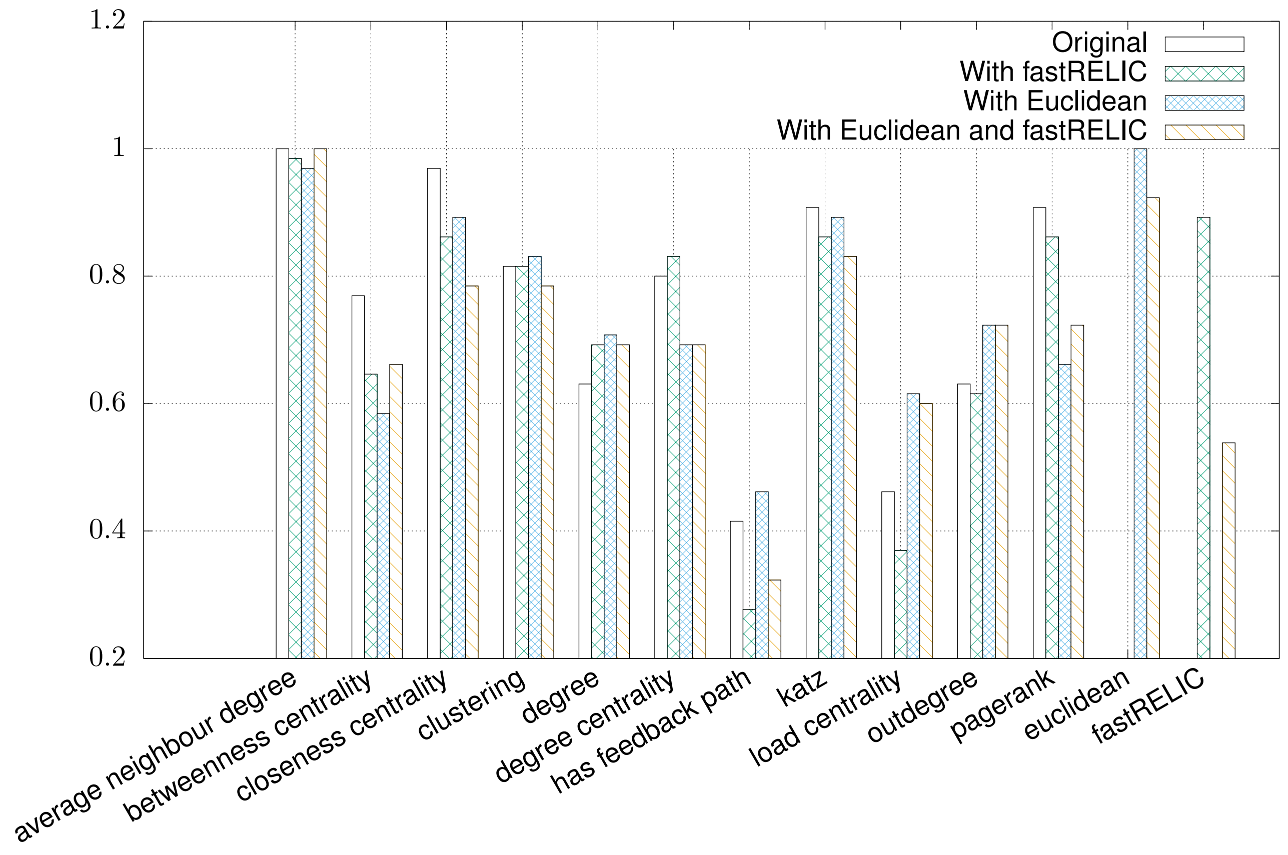


Figure 4.6 Feature Occurrence Percentage: Sequential Feature Selection

From Figure 4.6 there is support for the observations to the feature permutation methodology, which is the 2 most important primary features are "average neighbour degree" and "katz". Additionally, from Figure 4.6, across all implementations, the feature "has feedback path" does not appear more than 50% for all 5 runs. To test the importance of features occurring less than 50% for all 5 runs, those features are removed in all implementations respectively and re-tested for another 5 times.

From Table 4.7 and 4.8, the average of 5 run are almost identical to Table 4.3 and 4.4, with only a difference in less than 2%. This suggests that the respective removed feature(s), does not add any value to the machine learning process. Removing these feature(s) will therefore decrease the computation time required for the neural network. However, since the feature(s) removed were not the same for all implementation, based on Table 4.7 and 4.8, the conclusion that both "has feedback path" and "load centrality" features does not add any value to the neural network for all implementation cannot be drawn.

To confirm that both features do not add value to the neural network regardless of implementation for additional features, the same amount of features will have to be removed across all implementations and the process will have to be repeated. Previously, the files "With Euclidean" and the files "With Euclidean and fastRELIC", only had the feature "had feedback path" removed. Repeating the process for only these 2 implementation removing both the feature "has feedback path" and "load centrality", yields the results shown in Table 4.9 and 4.10.

File Name	Model Accuracy			
	Original	With fastRELIC	With Euclidean	With Euclidean and fastRELIC
b01_reset.csv	0.97	0.97	0.98	0.96
b02_reset.csv	0.91	0.94	0.94	0.90
b04_reset.csv	0.57	0.81	0.98	0.99
b06_reset.csv	1.00	1.00	1.00	1.00
b08_reset.csv	0.57	0.91	0.84	0.91
b09_reset.csv	0.63	0.79	0.65	0.73
b10_reset.csv	0.73	0.81	0.76	0.83
b14_reset.csv	0.56	0.68	0.61	0.66
completogpio.csv	0.84	0.88	0.86	0.87
FSM.csv	0.65	0.79	0.71	0.80
MEMORY_INTERFACE.csv	0.93	0.97	0.96	0.97
spi_axi_master.csv	0.98	0.98	0.96	0.91
uart.csv	0.44	0.72	0.55	0.70
average	0.75	0.86	0.83	0.86

Table 4.7 Model Accuracy

File Name	State Register Accuracy			
	Original	With fastRELIC	With Euclidean	With Euclidean and fastRELIC
b01_reset.csv	0.95	0.95	0.97	0.94
b02_reset.csv	0.88	0.92	0.91	0.87
b04_reset.csv	0.36	0.89	0.53	0.82
b06_reset.csv	0.99	1.00	0.99	1.00
b08_reset.csv	0.45	0.62	0.32	0.64
b09_reset.csv	0.86	0.95	0.97	0.96
b10_reset.csv	0.87	0.88	0.93	0.90
b14_reset.csv	0.59	0.21	0.28	0.22
completogpio.csv	0.49	0.98	1.00	1.00
FSM.csv	0.40	0.79	0.56	0.84
MEMORY_INTERFACE.csv	0.30	0.63	0.54	0.63
spi_axi_master.csv	0.32	0.50	0.71	0.68
uart.csv	0.24	0.75	0.45	0.68
average	0.59	0.77	0.70	0.78

Table 4.8 State Register Accuracy

File Name	Model Accuracy	
	With Euclidean	With Euclidean and fastRELIC
b01_reset.csv	0.99	0.97
b02_reset.csv	0.91	0.86
b04_reset.csv	0.98	0.99
b06_reset.csv	1.00	1.00
b08_reset.csv	0.82	0.91
b09_reset.csv	0.66	0.79
b10_reset.csv	0.79	0.85
b14_reset.csv	0.61	0.66
completogpio.csv	0.86	0.87
FSM.csv	0.71	0.80
MEMORY_INTERFACE.csv	0.96	0.96
spi_axi_master.csv	0.96	0.92
uart.csv	0.53	0.69
average	0.83	0.87

Table 4.9 Model Accuracy

File Name	State Register Accuracy	
	With Euclidean	With Euclidean and fastRELIC
completogpio.csv	0.98	0.95
b02_reset.csv	0.88	0.82
b04_reset.csv	0.55	0.81
b06_reset.csv	0.99	1.00
b08_reset.csv	0.27	0.63
b09_reset.csv	0.99	0.95
b10_reset.csv	0.95	0.89
b14_reset.csv	0.26	0.18
completogpio.csv	1.00	1.00
FSM.csv	0.56	0.88
MEMORY_INTERFACE.csv	0.53	0.62
spi_axi_master.csv	0.71	0.67
uart.csv	0.43	0.67
average	0.70	0.77

Table 4.10 State Register Accuracy

Conclusion for Sequential Feature Selection

Sequential Feature Selection has supported the observations from Feature Permutation, as well as gave insights to which Features are a hindrance/redundant. From Table 4.9 and 4.10, both the Model Accuracy and State Register Accuracy was minimally affected, even when both features "has feedback path" and "load centrality" have been removed. To further improve this results, further test with more features removed, using a higher threshold for percentage of feature selection can be used.

4.2.5 Implementation Selection

Based on the results obtained, the two best performing implementations are "With fastRELIC" and "With Euclidean and fastRELIC". As the implementation "With fastRELIC" has performed similarly well when compared to the implementation "With Euclidean and fastRELIC" with a feature lesser, the implementation "With fastRELIC" will be chosen for hyperparameter tuning.

4.3 Talos Hyperparameter Tuning

Briefly mentioned in section 4.1.1 was a method to optimize the neural network. The "Talos" module is used to automate the repeated training and testing process of the neural network, with a range of preset parameters. The most optimized model is determined by "Talos" based on a set metric. The model can then be deployed for predictions or further training.

The range of hyperparameters that is automated with "Talos" is shown in table 4.11:

Hyperparameter	Input
Optimizer	Adam, RMSprop
Loss Function	Binary Crossentropy, Sparse Categorical Crossentropy
Batch Size	10, 32
Epochs	1, 5, 10, 24
Dropout	0, 0.25, 0.5
First Neurons	20, to 80, increment of 4 per step
Hidden Layers	5, 15, 35
Hidden Neurons	7 to 490, increment of 7 per step
Hidden Layer Activation Function	ReLU, ELU
Last Activation Function	Sigmoid

Table 4.11 Talos Parameters

4.4 Tuning Model 1

The file used for testing has been selected to be "b14_reset.csv". This is due to the file containing near 20% of all registers. By default, "Talos" uses grid search to process and find the most optimal hyperparameter however, due to the large number of permutations, high number of hidden layers and hidden neurons, using grid search resulted in a really slow process. Therefore, random search of 20% of the permutations with a probabilistic reduction using the metric "forest" was used to find the most optimal hyperparameters. Two of the hyperparameter permutation which resulted in the highest accuracy was then chosen shown in 4.12, represented as Model 1 and Model 2 .

From the hyperparameter tuning, the optimal hyperparameters from Talos is shown in table 4.12

Hyperparameter	Model 1	Model 2
Optimizer	adam	adam
Loss Function	Sparse Categorical Crossentropy	Binary Crossentropy
Batch Size	32	32
Epochs	1	10
Dropout	0.25	0
First Neurons	35	50
Hidden Layers	5	35
Hidden Neurons	7	287
Hidden Layer Activation Function	ReLU	ReLu
Last Activation Function	Sigmoid	Sigmoid

Table 4.12 Optimal Hyperparameters (b14_reset)

Deploying Model 1

Recreating these 2 models, the models were evaluated for accuracy and state register accuracy for a total of 500 times per file and the average accuracy across all files was then calculated. The results are shown in Table 4.13.

	Model Accuracy		State Register Accuracy	
File Name	Model 1	Model 2	Model 1	Model 2
b01_reset.csv	0.71	0.62	0.54	0.38
b02_reset.csv	0.63	0.54	0.53	0.40
b04_reset.csv	0.95	0.92	0.44	0.43
b06_reset.csv	0.80	0.76	0.59	0.43
b08_reset.csv	0.83	0.81	0.39	0.27
b09_reset.csv	0.81	0.84	0.63	0.43
b10_reset.csv	0.78	0.77	0.62	0.39
b14_reset.csv	0.84	0.82	0.66	0.48
completogpio.csv	0.87	0.83	0.45	0.36
FSM.csv	0.68	0.63	0.49	0.35
MEMORY_INTERFACE.csv	0.92	0.93	0.15	0.24
spi_axi_master.csv	0.98	0.96	0.04	0.32
uart.csv	0.27	0.33	0.01	0.11
average	0.77	0.75	0.43	0.35

Table 4.13 Optimized Model Accuracy (b14_reset)

From Table 4.13 the accuracy was high, yet the state register accuracy remains very low for both models. From these results, it can be observed that this model performed worse than the pre-optimized neural network. The following conclusions can therefore be drawn:

- Model resulted in high false negative rate
- b14_reset.csv is not suitable for testing due to the low amount of state registers in the file and due to the file containing valuable information for training.
- There is no or low correlation between Model Accuracy and State Register Accuracy

4.5 Tuning Model 2

Due to the poor results produced in section 4.4 the tuning process was repeated using "uart.csv" instead. "uart.csv", though it contains less registers, it has an even spread of state register and non-state registers. This makes it ideal for testing the neural network during the tuning process. Similarly, 2 best performing hyperparameter permutations were recreated and tested for 500 times.

Hyperparameter	Model 1	Model 2
Optimizer	adam	RMSprop
Loss Function	Sparse Categorical Crossentropy	Binary Crossentropy
Batch Size	32	10
Epochs	5	10
Dropout	0	0.5
First Neurons	20	20
Hidden Layers	15	15
Hidden Neurons	7	217
Hidden Layer Activation Function	ReLU	ELU
Last Activation Function	Sigmoid	Sigmoid

Table 4.14 Optimal Hyperparameters (uart)

Deploying Model 2

The results are shown in table

	Model Accuracy		State Register Accuracy	
File Name	Model 1	Model 2	Model 1	Model 2
b01_reset.csv	0.41	0.80	0.02	0.98
b02_reset.csv	0.28	0.86	0.04	0.98
b04_reset.csv	0.97	0.66	0.02	0.96
b06_reset.csv	0.63	0.55	0.04	0.99
b08_reset.csv	0.77	0.44	0.03	0.98
b09_reset.csv	0.92	0.36	0.04	0.99
b10_reset.csv	0.77	0.55	0.03	0.99
b14_reset.csv	0.99	0.56	0.05	0.99
completogpio.csv	0.79	0.66	0.01	0.97
FSM.csv	0.54	0.55	0.02	0.99
MEMORY_INTERFACE.csv	0.91	0.97	0.00	0.69
spi_axi_master.csv	0.99	0.87	0.00	0.65
uart.csv	0.27	0.63	0.00	0.60
average	0.71	0.65	0.02	0.91

Table 4.15 Optimized Model Accuracy (uart)

It can be seen from Table 4.15 that though Model 1 has significantly higher model accuracy compared to Model 2, the state register accuracy is significantly lower. This suggests that Model 1 produces a significant amount of false negative, while Model 2 has a lower Model Accuracy, but significantly higher State Register Accuracy, out performing the pre-optimized model in most of the files. This suggests that Model 2 might be underfitted due to low model accuracy, thus producing a high amount of false positive.

To improve this result, the amount of epoch is raised from 10 to 22 shown in Table 4.16, with other hyperparameters kept the same.

Hyperparameter	Model 2
Optimizer	RMSprop
Loss Function	Binary Crossentropy
Batch Size	10
Epochs	22
Dropout	0.5
First Neurons	20
Hidden Layers	15
Hidden Neurons	217
Hidden Layer Activation Function	ELU
Last Activation Function	Sigmoid

Table 4.16 Modified Optimal Hyperparameters (uart)

The results is shown in table 4.17

File Name	Model Accuracy	State Register Accuracy
b01_reset.csv	0.95	0.99
b02_reset.csv	0.93	0.99
b04_reset.csv	0.69	0.94
b06_reset.csv	0.83	1.00
b08_reset.csv	0.73	0.97
b09_reset.csv	0.56	0.99
b10_reset.csv	0.67	1.00
b14_reset.csv	0.53	1.00
completogpio.csv	0.81	0.96
FSM.csv	0.64	0.98
MEMORY_INTERFACE.csv	0.99	0.85
spi_axi_master.csv	0.85	0.66
uart.csv	0.56	0.48
average	0.75	0.91

Table 4.17 Modified Optimized Model Accuracy (uart)

4.6 Conclusion for Tuning Neural Network

By increasing the epoch from 10 to 22 shown in 4.17, the average model accuracy has increased by a significant 10%, while keeping the average State Register Accuracy the same. This increment implies a significantly lesser false positive on average compared to the original optimal hyperparameters shown in 4.15 (Model 2).

With an overall increase per file in Model Accuracy, it is worth noting that not all per file Model Accuracy is high even if there is an overall increase. This is especially true when observing the results in Table 4.17 for the file "uart.csv". "uart.csv" is the worst performing file with a significant decrease in both Model Accuracy and State Register Accuracy after changing the epoch, thus showing this neural network can still be further optimized.

Comparing the implementation "With fastRELIC" results of the tuned model in Table 4.17 to the results of the pre-optimize model in Table 4.7 and Table 4.8, the Model Accuracy has decrease, while the State Register Accuracy has increased. Though it is preferred to have both accuracy being equivalently high, there is always a trade off between these 2 metrics. However, the Model chosen should be able to produce results with a higher State Register Accuracy. A low State Register Accuracy will result in FSM extraction process impossible, while a low State Model Accuracy will result in extraction of FSM process being more difficult.

To conclude, even though the model presents a high average Model Accuracy and high average State Register Accuracy, the model does not produce a consistently high Model Accuracy and State Register Accuracy on a per file per run basis. However, with such a high accuracy on average for both metrics, this proves that Machine Learning is well suited for this application, given the right features.

5 Conclusion and Future Work

5.1 Future Work

5.1.1 The Data

A larger data set can be trial with, ensuring a good spread of State and Non-State Registers in both training and testing files.

5.1.2 Pre-Processing: Feature Selection

Based on the results produced, the correlation between each feature can be further looked into, filtering out even more features that will hinder the neural network. Heat maps can be used and the effects of removing highly correlated features should be studied.

Though computationally more expensive, exhaustive feature selection can be tested out instead of sequential feature selection.

The counting algorithm used to implement fastRELIC as an additional feature, results in high performance in this experiment. However, the scores are inconsistent as the count is normalized per file. Therefore the fastRELIC result has a high dependency on the amount of registers in a file. Thus using Machine Learning to assign similarity scores to replace the reliance on fastRELIC can be explored. These 2 machine learning models can then be integrated, simplifying the entire process.

5.1.3 Neural Network Architecture and Tuning

Multiple different types of neural network architecture can be tested out, with varying neurons and activation functions per layer. Instead of compiling all the training data to 1 file which is then used to train the neural network, training the neural network progressively can also be tested out with differing hyperparameters per training file.

5.2 Conclusion

Comparing the results discussed in Chapter 4 to the results obtained solely by the method of fastRELIC, the results of this thesis shows that State Register identification using machine learning is a viable alternative which compliments the fastRELIC method, achieving an average high Model Accuracy and State Register Accuracy. This thesis should therefore be viewed as an evolution to the fastRELIC method. However, due to the experiments only being trial on a limited data set, which achieved inconsistent results on a single run, the model must still be regarded to be in its infancy and not ready to be deployed for use. Therefore, to be able to use the model for State Register identification in the near future, further exploration discussed in Section 5.1 has to be explored to ensure the reliability of the model.

A Supplementary material

Bibliography

[1] J. Baehr, A. Bernardini, G. Sigl, and U. Schlichtmann, “Machine learning and structural characteristics for reverse engineering,” *Integration*, vol. 72, pp. 1–12, 2020.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *40th IEEE Symposium on Security and Privacy (S&P’19)*, 2019.

[4] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, “Gate-level netlist reverse engineering for hardware security: Control logic register identification,” in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 1334–1337.

[5] M. Brunner, J. Baehr, and G. Sigl, “Improving on state register identification in sequential hardware reverse engineering,” in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 151–160.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[7] R. Shaikh, “Feature selection techniques in machine learning with python,” https://towardsdatascience.com/feature-selection-techniques-in-machine-learning-with-python-f24e7da3f36e, 2018.

[8] S. Raschka, “Mlxtend: Providing machine learning and data science utilities and extensions to python’s scientific computing stack,” *The Journal of Open Source Software*, vol. 3, no. 24, Apr. 2018. [Online]. Available: http://joss.theoj.org/papers/10.21105/joss.00638

[9] D. A. S. Aric A. Hagberg and P. J. Swart, “Exploring network structure, dynamics, and function using networkx,” in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, 2008, pp. 11–15.

[10] F. SHAIKH, “Why are gpus necessary for training deep learning models?” https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/, 2017.

[11] F. Bre, J. Gimenez, and V. Fachinotti, “Prediction of wind pressure coefficients on building surfaces using artificial neural networks,” *Energy and Buildings*, vol. 158, 11 2017.

[12] Wes McKinney, “Data Structures for Statistical Computing in Python,” in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[13] F. Chollet *et al.*, “Keras,” https://keras.io, 2015.

[14] “Autonomio talos [computer software],” http://github.com/autonomio/talos, 2019.